Message Passing

Concepts:
- Synchronous message passing - channel
- Asynchronous message passing - port
  - Send and receive / selective receive
  - Rendezvous bidirectional comms - entry
  - Call and accept ... reply

Models:
- Channel: relabelling, choice & guards
- Port: message queue, choice & guards
- Entry: port & channel

Practice:
- Distributed computing (disjoint memory)
- Threads and monitors (shared memory)

---

10.1 Synchronous Message Passing - channel

A sender communicates with a receiver using a single channel.

The sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.

```java
Channel chan = new Channel();
tx.start(new Sender(chan, senddisp));
rx.start(new Receiver(chan, recvdisp));
```

Instances of `ThreadPanel`

Instances of `SlotCanvas`

---

Synchronous message passing - applet

A sender communicates with a receiver using a single channel.

The sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.

```java
Channel chan = new Channel();
tx.start(new Sender(chan, senddisp));
rx.start(new Receiver(chan, recvdisp));
```

Instances of `ThreadPanel`

Instances of `SlotCanvas`
The implementation of Channel is a monitor that has synchronized access methods for send and receive.

```java
class Channel extends Selectable {
    Object chann = null;

    public synchronized void send(Object v) throws InterruptedException {
        chann = v;
        signal();
        while (chann != null) wait();
    }

    public synchronized Object receive() throws InterruptedException {
        block(); clearReady(); // part of Selectable
        Object tmp = chann; chann = null; notifyAll(); // could be notify()
        return(tmp);
    }
}
```

Selectable is described later.

```java
class Sender implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Sender(Channel c, SlotCanvas d) {chan=c; display=d;}
    public void run() {
        try { int ei = 0;
            while(true) {
                display.enter(String.valueOf(ei));
                ThreadPanel.rotate(12);
                chan.send(new Integer(ei));
                display.leave(String.valueOf(ei));
                ei=(ei+1)%10; ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```

```java
class Receiver implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Receiver(Channel c, SlotCanvas d) {chan=c; display=d;}
    public void run() {
        try { Integer v=null;
            while(true) {
                ThreadPanel.rotate(180);
                if(v!=null) display.leave(v.toString());
                v = (Integer)chan.receive();
                display.enter(v.toString());
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e){}
    }
}
```

Concurrent Java implementation of channel send and receive.

Model

```
range M = 0..9 // messages with values up to 9
SENDER = SENDER[0], // shared channel chan
SENDER[e:M] = (chan.send[e]-> SENDER[(e+1)%10]).
RECEIVER = (chan.receive[v:M]-> RECEIVER).
// relabeling to model synchronization
||SyncMsg = (SENDER || RECEIVER)
/{chan/chan.{send,receive}}. LTS?
```

How can this be modelled directly without the need for relabeling?

<table>
<thead>
<tr>
<th>message operation</th>
<th>FSP model</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(c,chan)</td>
<td>?</td>
</tr>
<tr>
<td>v = receive(chan)</td>
<td>?</td>
</tr>
</tbody>
</table>
Concurrency: message passing

How should we deal with multiple channels?

Sender[n]
send(e,c)

Channels

How would we model this in FSP?

Select
when G1 and v1=receive(chan1) => S1;
or
when G2 and v2=receive(chan2) => S2;
or
when Gn and vn=receive(chan_n) => Sn;
end

Select statement...

Java implementation - selective receive

```java
class MsgCarPark implements Runnable {
    private Channel arrive, depart;
    private int spaces, N;
    private StringCanvas disp;

    public MsgCarPark(Channel a, Channel l, StringCanvas d, int capacity) {
        depart=l; arrive=a; N=spaces=capacity; disp=d;
    }

    public void run() {
        try {
            Select sel = new Select();
            sel.add(depart); sel.add(arrive);
            while(true) {
                ThreadPanel.rotate(12);
                arrive.guard(spaces>0); depart.guard(spaces<N);
                switch (sel.choose()) {
                    case 1: depart.receive(); display(++spaces); break;
                    case 2: arrive.receive(); display(--spaces); break;
                    ...
                }
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Java implementation - selective receive

```java
public void run() {
    try {
        Select sel = new Select();
        sel.add(depart);
        sel.add(arrive);
        while(true) {
            ThreadPanel.rotate(12);
            arrive.guard(spaces>0);
            depart.guard(spaces<N);
            switch (sel.choose()) {
                case 1: depart.receive(); display(++spaces); break;
                case 2: arrive.receive(); display(--spaces); break;
                ...
            }
        }
    } catch (InterruptedException e) {
    }
}
```
10.2 Asynchronous Message Passing - port

- send(e, c) - send the value of the expression e to port p. The process calling the send operation is not blocked. The message is queued at the port if the receiver is not waiting.

- v = receive(c) - receive a value into local variable v from port p. The process calling the receive operation is blocked if there are no messages queued to the port.

Asynchronous message passing - applet

Two senders communicate with a receiver via an "unbounded" port.
Each sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.

Java implementation - port

```java
class Port extends Selectable {
    Vector queue = new Vector();
    public synchronized void send(Object v) {
        queue.addElement(v);
        signal();
    }
    public synchronized Object receive() throws InterruptedException {
        block(); clearReady();
        Object tmp = queue.elementAt(0);
        queue.removeElementAt(0);
        return(tmp);
    }
}
```

Instances of ThreadPanel
Instances of SlotCanvas

Port model

```plaintext
range M = 0..9 // messages with values up to 9
set S = {[M], [M][M]} // queue of up to three messages
PORT = \{(send[x:M] -> PORT[x])\} // empty state, only send permitted
PORT[h:M] = \{(send[x:M] -> PORT[x][h])
| receive[h] -> PORT\} // one message queued to port
PORT[t:S][h:M] = \{(send[x:M] -> PORT[x][t][h])
| receive[h] -> PORT[t]\} // two or more messages queued to port

LTS? // minimise to see result of abstracting from data values
| | APORT = PORT/{send/send[M], receive/receive[M].
```
Concurrency: message passing

**Safety?**

Rendezvous - entry

Rendezvous is a form of request-reply to support client server communication. Many clients may request service, but only one is serviced at a time.

AsynchMsg = (s[1..2]:ASENDER || ARECEIVER || port:PORT) / {s[1..2].port.send/port.send}.

**Safety?**

Rendezvous

- **res**=call(e,req) - send the value **req** as a request message which is queued to the entry **e**.
- **req**=accept(e) - receive the value of the request message from the entry **e** into local variable **req**. The calling process is **blocked** if there are no messages queued to the entry.
- **reply(e,res)** - send the value **res** as a reply message to entry **e**.

asynchronous message passing - applet

Two clients call a server which services a request at a time.

Instances of ThreadPanel

Instances of SlotCanvas

Entry entry = new Entry();
c1A.start(new Client(entry,clientAdisp, "A"));
c1B.start(new Client(entry,clientBdisp, "B"));
sv.start(new Server(entry,serverdisp));
Entries are implemented as extensions of ports, thereby supporting queuing and selective receipt.

The call method creates a channel object on which to receive the reply message. It constructs and sends to the entry a message consisting of a reference to this channel and a reference to the req object. It then awaits the reply on the channel.

The accept method keeps a copy of the reply channel reference; the reply method sends the reply message to this channel.

Entries are implemented as extensions of ports, thereby supporting queuing and selective receipt.

Do call, accept and reply need to be synchronized methods?

We reuse the models for ports and channels...

set M = {replyA, replyB}    // reply channels
|| ENTRY = PORT/{call/send, accept/receive}.
CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT).
SERVER = (entry.accept[ch:M]->[ch]->SERVER).
|| | EntryDemo = (CLIENT('replyA) || CLIENT('replyB)
| | entry:ENTRY || SERVER ).
Summary

◆ Concepts
  - synchronous message passing - channel
  - asynchronous message passing - port
    - send and receive / selective receive
  - rendezvous bidirectional comms - entry
    - call and accept ... reply

◆ Models
  - channel: relabelling, choice & guards
  - port: message queue, choice & guards
  - entry: port & channel

◆ Practice
  - distributed computing (disjoint memory)
  - threads and monitors (shared memory)

Course Outline

♦ Processes and Threads
♦ Concurrent Execution
♦ Shared Objects & Interference
♦ Monitors & Condition Synchronization
♦ Deadlock
♦ Safety and Liveness Properties
♦ Model-based Design
♦ Dynamic systems
♦ Concurrent Software Architectures
♦ Message Passing
♦ Timed Systems